Chapitre 29

Initialisation d'un serveur

Nous avons vu au chapitre 6 que l'initialisation d'un serveur s'effectue grâce à l'appel système :

```
bind(socket, adresse, taille);
```

Nous allons étudier l'implémentation de cet appel système dans ce chapitre, dans sa partie générale, dans le cas de la couche réseau IPv4 et de la couche de transport UDP.

29.1 Partie générale

29.1.1 Fonction d'appel

Code Linux 2.6.10

La fonction d'appel `sys_bind()` est définie dans le fichier `linux/net/socket.c`:

```
1275 /*
1276  *      Lie un nom a une socket. Rien de plus a faire ici puisque c'est de
1277  *      la responsabilite du protocole de traiter l'adresse locale.
1278  *
1279  *      Nous deplacons l'adresse de la socket depuis l'espace noyau avant d'appeler
1280  *      la couche de protocole (ayant egalement verifie que l'adresse est ok).
1281  */
1282
1283 asmlinkage long sys_bind(int fd, struct sockaddr __user *umyaddr, int addrlen)
1284 {
1285         struct socket *sock;
1286         char address[MAX_SOCK_ADDR];
1287         int err;
1288
1289         if((sock = sockfd_lookup(fd,&err))!=NULL)
1290         {
1291                 if((err=move_addr_to_kernel(umyaddr,addrlen,address))>=0) {
1292                         err = security_socket_bind(sock, (struct sockaddr *)address,
                                                           addrlen);
1293                         if (err) {
1294                                 sockfd_put(sock);
1295                                 return err;
1296                         }
1297                         err = sock->ops->bind(sock, (struct sockaddr *)address, addrlen);
1298                 }
1299                 sockfd_put(sock);
1300         }
1301         return err;
1302 }
```

Autrement dit :

- On déclare un descripteur de socket, une adresse et une variable pour la valeur renvoyée.
- On essaie d'instantier le descripteur de socket avec celui associé au numéro de fichier (de socket) passé en argument. Si on n'y parvient pas, on renvoie l'opposé du code d'erreur fourni par la fonction `sockfd_lookup()`.
- L'adresse du serveur se trouvant dans l'espace utilisateur, on essaie de la rapatrier dans l'espace noyau, grâce à la fonction `move_addr_to_kernel()` étudiée ci-après. Si on n'y parvient pas, on libère le descripteur de socket et on renvoie l'opposé du code d'erreur fourni par cette dernière fonction.
- On fait appel à la fonction de sécurité `security_socket_bind()`, dont nous ne nous occuperons pas ici. Si elle n'accorde pas le droit d'accès, on libère le descripteur de fichier associé à la socket et on renvoie le code d'erreur fourni par cette fonction.
- On fait appel à la fonction spécifique associée à la famille de protocoles, on libère le descripteur de fichier associé au descripteur de socket et on renvoie le code fourni par la fonction spécifique.

29.1.2 Passage espace utilisateur/espace noyau

Code Linux 2.6.10

La fonction `move_addr_to_kernel()` est définie dans le fichier `linux/net/socket.c`:

```
210 /**
```

```
211  *      move_addr_to_kernel     -       copie une adresse de socket dans l'espace noyau
212  *      @uaddr : Adresse dans l'espace utilisateur
213  *      @kaddr : Adresse dans l'espace noyau
214  *      @ulen  : Longueur dans l'espace utilisateur
215  *
216  *      L'adresse est copiee dans l'espace noyau. Si l'adresse fournie est
217  *      trop longue, un code d'erreur de -EINVAL est renvoye. Si la copie donne
218  *      des adresses non valides, -EFAULT est renvoye. En cas de succes, 0 est renvoye.
219  */
220
221 int move_addr_to_kernel(void __user *uaddr, int ulen, void *kaddr)
222 {
223         if(ulen<0||ulen>MAX_SOCK_ADDR)
224                 return -EINVAL;
225         if(ulen==0)
226                 return 0;
227         if(copy_from_user(kaddr,uaddr,ulen))
228                 return -EFAULT;
229         return 0;
230 }
```

Le code se comprend aisément.

29.2 Cas de IPv4

29.2.1 Fonction spécifique d'initialisation du serveur

Nous avons vu, à propos de `inet_dgram_ops` pour UDP, que la fonction d'initialisation du serveur spécifique à IPv4 s'appelle `inet_bind()`. Cette fonction est définie dans le fichier `linux/net/ipv4/af_inet.c` :

Code Linux 2.6.10

```
385 /* Elle est desactivee par defaut, voir ci-dessous. */
386 int sysctl_ip_nonlocal_bind;
387
388 int inet_bind(struct socket *sock, struct sockaddr *uaddr, int addr_len)
389 {
390         struct sockaddr_in *addr = (struct sockaddr_in *)uaddr;
391         struct sock *sk = sock->sk;
392         struct inet_opt *inet = inet_sk(sk);
393         unsigned short snum;
394         int chk_addr_ret;
395         int err;
396
397         /* Si la socket a sa propre fonction bind, utilisons-la. (RAW) */
398         if (sk->sk_prot->bind) {
399                 err = sk->sk_prot->bind(sk, uaddr, addr_len);
400                 goto out;
401         }
402         err = -EINVAL;
403         if (addr_len < sizeof(struct sockaddr_in))
404                 goto out;
405
406         chk_addr_ret = inet_addr_type(addr->sin_addr.s_addr);
407
408         /* Specifie par aucun standard en soi, cependant elle brise beaucoup trop
409          * d'applications lorsque supprime. C'est dommage puisque
410          * permettre aux applications d'effectuer une liaison non locale resout
411          * certains problemes avec les systemes utilisant un adressage dynamique.
412          * (c'est-a-dire que vos serveurs demarreront meme si votre lien ISDN
413          *  est temporairement inactif)
414          */
```

```
415         err = -EADDRNOTAVAIL;
416         if (!sysctl_ip_nonlocal_bind &&
417             !inet->freebind &&
418             addr->sin_addr.s_addr != INADDR_ANY &&
419             chk_addr_ret != RTN_LOCAL &&
420             chk_addr_ret != RTN_MULTICAST &&
421             chk_addr_ret != RTN_BROADCAST)
422                 goto out;
423 
424         snum = ntohs(addr->sin_port);
425         err = -EACCES;
426         if (snum && snum < PROT_SOCK && !capable(CAP_NET_BIND_SERVICE))
427                 goto out;
428 
429         /*      Nous gardons une paire d'adresses. rcv_saddr est celle utilisee par
430          *      les consultations de hachage et saddr est utilisee pour la transmission.
431          *
432          *      Dans l'API BSD, ce sont les memes sauf lorsqu'il
433          *      serait illegal de les utiliser (multicast/broadcast), auquel
434          *      cas l'adresse du peripherique d'envoi est utilise.
435          */
436         lock_sock(sk);
437 
438         /* Verifie ces erreurs (socket active, double liaison). */
439         err = -EINVAL;
440         if (sk->sk_state != TCP_CLOSE || inet->num)
441                 goto out_release_sock;
442 
443         inet->rcv_saddr = inet->saddr = addr->sin_addr.s_addr;
444         if (chk_addr_ret == RTN_MULTICAST || chk_addr_ret == RTN_BROADCAST)
445                 inet->saddr = 0;  /* Utiliser le peripherique */
446 
447         /* S'assurer que nous avons le droit de se relier ici. */
448         if (sk->sk_prot->get_port(sk, snum)) {
449                 inet->saddr = inet->rcv_saddr = 0;
450                 err = -EADDRINUSE;
451                 goto out_release_sock;
452         }
453 
454         if (inet->rcv_saddr)
455                 sk->sk_userlocks |= SOCK_BINDADDR_LOCK;
456         if (snum)
457                 sk->sk_userlocks |= SOCK_BINDPORT_LOCK;
458         inet->sport = htons(inet->num);
459         inet->daddr = 0;
460         inet->dport = 0;
461         sk_dst_reset(sk);
462         err = 0;
463 out_release_sock:
464         release_sock(sk);
465 out:
466         return err;
467 }
```

Autrement dit :

- On déclare une adresse Internet, que l'on instantie avec celle passée en argument.
- On déclare un descripteur de couche transport, que l'on instantie avec celui associé au descripteur de socket passé en argument.
- On déclare une option Internet, que l'on instantie avec celle associée au descripteur de transport.

- Si la socket possède sa propre fonction d'initialisation (ce qui est le cas du mode brut mais pas de UDP, comme le montre `udp_prot`, ni de TCP, comme le montre `tcp_prot`), on fait appel à celle-ci, on libère le descripteur de couche transport et on renvoie le code fourni par cette fonction.
- Si la longueur de l'adresse, telle qu'entrée en argument, est inférieure à la longueur d'une adresse standard d'Internet, on renvoie l'opposé du code d'erreur `EINVAL`.
- On détermine le type d'adresse grâce à la fonction auxiliaire `inet_addr_type()`, qui sera étudiée ci-dessous. La valeur de retour est l'une des trois constantes symboliques `RTN_UNICAST`, `RTN_BROADCAST`, `RTN_MULTICAST` ou `RTN_LOCAL`.
- Dans un certain nombre de cas choisis par Linux, on renvoie l'opposé du code d'erreur `EADDRNOTAVAIL`.
- On détermine le numéro de port source `snum` à partir de l'adresse spécifiée.
- Si le numéro de port spécifié ne convient pas, par exemple s'il est inférieur à 1 024 alors que le processus n'appartient pas au super-utilisateur, on renvoie l'opposé du code d'erreur `EACCES`.

  La constante `PROT_SOCK` est définie dans le fichier en-tête `linux/include/net/sock.h`: Code Linux 2.6.10

```
608 /* Les sockets 0-1023 ne peuvent pas etre liees a moins que vous ne soyez le
       super-utilisateur */
609 #define PROT_SOCK       1024
```

- On verrouille le descripteur de couche transport de façon à ce que personne d'autre n'intervienne sur lui.
- Si la socket est déjà active ou si l'on essaie d'y connecter un second serveur, on libère le descripteur de couche transport et on renvoie l'opposé du code d'erreur `EINVAL`.
- On initialise l'adresse de réception de l'option internet associée au descripteur de couche transport, ce qui est le but principal de cette fonction.
- Dans le cas d'une multidiffusion ou de la diffusion générale, on positionne cette adresse à zéro, ce qui permet d'indiquer au périphérique réseau que l'on veut un tel type de diffusion.
- On vérifie que l'on peut utiliser le numéro de port spécifié. Si ce n'est pas le cas, on remet à zéro les adresses précédentes, on libère le descripteur de couche transport et on renvoie l'opposé du code d'erreur `EADDRINUSE`. On utilise pour cela la fonction `get_port()` spécifique au protocole de transport.
- On renseigne encore quelques champs du descripteur de couche transport et de l'option internet, on libère le descripteur et on renvoie 0.

29.2.2 Détermination du type d'adresse

La fonction `inet_addr_type()` est définie dans le fichier `linux/net/ipv4/fib_frontend.c`: Code Linux 2.6.10

```
127 unsigned inet_addr_type(u32 addr)
128 {
129         struct flowi            fl = { .nl_u = { .ip4_u = { .daddr = addr } } };
130         struct fib_result       res;
131         unsigned ret = RTN_BROADCAST;
132
133         if (ZERONET(addr) || BADCLASS(addr))
134                 return RTN_BROADCAST;
135         if (MULTICAST(addr))
136                 return RTN_MULTICAST;
137
138 #ifdef CONFIG_IP_MULTIPLE_TABLES
139         res.r = NULL;
```

```
140 #endif
141
142         if (ip_fib_local_table) {
143                 ret = RTN_UNICAST;
144                 if (!ip_fib_local_table->tb_lookup(ip_fib_local_table,
145                                                    &fl, &res)) {
146                         ret = res.type;
147                         fib_res_put(&res);
148                 }
149         }
150         return ret;
151 }
```

Autrement dit :

- On déclare un flux Internet, que l'on instantie.
- On déclare un résultat de consultation de table de routage et une valeur de retour.
- Si l'adresse passée en argument est nulle ou correspond à une mauvaise classe d'adresses, on renvoie `RTN_BROADCAST`.
- S'il s'agit d'une adresse de multidiffusion, on renvoie `RTN_MULTICAST`.
- Si la table de routage locale n'est pas définie, on renvoie `RTN_BROADCAST`.
- Si la table de routage locale est définie :
  - On la consulte. Si on obtient un résultat, on place la réponse et on renvoie le type fourni par la fonction de consultation.

    La fonction en ligne `fib_res_put()` est définie dans le fichier en-tête `linux/include/net/ip_fib.h` :

Code Linux 2.6.10

```
260 static inline void fib_res_put(struct fib_result *res)
261 {
262         if (res->fi)
263                 fib_info_put(res->fi);
264 #ifdef CONFIG_IP_MULTIPLE_TABLES
265         if (res->r)
266                 fib_rule_put(res->r);
267 #endif
268 }
```

  - Sinon on renvoie la valeur `RTN_UNICAST`.

29.3 Obtention et vérification du port dans le cas UDP

Rappelons que le numéro de port en cas d'émission par un client est attribué dynamiquement pour savoir à quel processus remettre le paquet répondant à un paquet envoyé. Rappelons également que les numéros 0 à 1 023 sont réservés aux services connus.

Le tableau `sysctl_local_port_range[]` permet de savoir dans quel intervalle le système peut attribuer ces numéros. Il est défini dans le fichier `linux/net/ipv4/tcp_ipv4.c` :

Code Linux 2.6.10

```
99  /*
100  * Ce tableau contient le premier et le dernier numeros de port locaux.
101  * Pour des systemes a tres grand usage, changer ce sysctl par
102  * 32768-61000
103  */
104 int sysctl_local_port_range[2] = { 1024, 4999 };
```

La fonction `get_port()` d'obtention et de vérification du numéro de port spécifique à IPv4/-UDP s'appelle `udp_v4_get_port()`, comme le montre `udp_prot`. Elle est définie dans le fichier `linux/net/ipv4/udp.c`:

Code Linux 2.6.10

```
120 /* Partage par udp v4/v6. */
121 int udp_port_rover;
122
123 static int udp_v4_get_port(struct sock *sk, unsigned short snum)
124 {
125         struct hlist_node *node;
126         struct sock *sk2;
127         struct inet_opt *inet = inet_sk(sk);
128
129         write_lock_bh(&udp_hash_lock);
130         if (snum == 0) {
131                 int best_size_so_far, best, result, i;
132
133                 if (udp_port_rover > sysctl_local_port_range[1] ||
134                     udp_port_rover < sysctl_local_port_range[0])
135                         udp_port_rover = sysctl_local_port_range[0];
136                 best_size_so_far = 32767;
137                 best = result = udp_port_rover;
138                 for (i = 0; i < UDP_HTABLE_SIZE; i++, result++) {
139                         struct hlist_head *list;
140                         int size;
141
142                         list = &udp_hash[result & (UDP_HTABLE_SIZE - 1)];
143                         if (hlist_empty(list)) {
144                                 if (result > sysctl_local_port_range[1])
145                                         result = sysctl_local_port_range[0] +
146                                                 ((result - sysctl_local_port_range[0]) &
147                                                  (UDP_HTABLE_SIZE - 1));
148                                 goto gotit;
149                         }
150                         size = 0;
151                         sk_for_each(sk2, node, list)
152                                 if (++size >= best_size_so_far)
153                                         goto next;
154                         best_size_so_far = size;
155                         best = result;
156                 next:;
157                 }
158                 result = best;
159                 for(i = 0; i < (1 << 16) / UDP_HTABLE_SIZE; i++, result += UDP_HTABLE_SIZE) {
160                         if (result > sysctl_local_port_range[1])
161                                 result = sysctl_local_port_range[0]
162                                         + ((result - sysctl_local_port_range[0]) &
163                                            (UDP_HTABLE_SIZE - 1));
164                         if (!udp_lport_inuse(result))
165                                 break;
166                 }
167                 if (i >= (1 << 16) / UDP_HTABLE_SIZE)
168                         goto fail;
169 gotit:
170                 udp_port_rover = snum = result;
171         } else {
172                 sk_for_each(sk2, node,
173                             &udp_hash[snum & (UDP_HTABLE_SIZE - 1)]) {
174                         struct inet_opt *inet2 = inet_sk(sk2);
175
176                         if (inet2->num == snum &&
177                             sk2 != sk &&
178                             !ipv6_only_sock(sk2) &&
```

```
179                             (!sk2->sk_bound_dev_if ||
180                              !sk->sk_bound_dev_if ||
181                              sk2->sk_bound_dev_if == sk->sk_bound_dev_if) &&
182                             (!inet2->rcv_saddr ||
183                              !inet->rcv_saddr ||
184                              inet2->rcv_saddr == inet->rcv_saddr) &&
185                             (!sk2->sk_reuse || !sk->sk_reuse))
186                                 goto fail;
187                 }
188         }
189         inet->num = snum;
190         if (sk_unhashed(sk)) {
191                 struct hlist_head *h = &udp_hash[snum & (UDP_HTABLE_SIZE - 1)];
192
193                 sk_add_node(sk, h);
194                 sock_prot_inc_use(sk->sk_prot);
195         }
196         write_unlock_bh(&udp_hash_lock);
197         return 0;
198
199 fail:
200         write_unlock_bh(&udp_hash_lock);
201         return 1;
202 }
```

Autrement dit :

- La variable globale `udp_port_rover`, définie à la ligne 121, permet de savoir où on en est dans la distribution des numéros de port locaux.
- On déclare une liste de descripteurs de nœuds d'information.
- On déclare un descripteur de couche transport distant `sk2`.
- On déclare une option internet, que l'on instantie avec celle associée au descripteur de couche transport local (celui passé en argument).
- On verrouille en écriture.
- Si le numéro de port source passé en argument est nul, c'est qu'il faut attribuer un numéro de port dynamiquement. Dans ce cas :
  - Si la variable `udp_port_rover` est sortie de l'intervalle dans lequel elle doit prendre ses valeurs, on lui donne comme valeur l'origine de cet intervalle.
  - On donne provisoirement comme numéro de port la valeur de cette variable, puis on utilise la procédure de hachage pour lui donner sa valeur définitive à partir de celle-ci, que l'on attribue également comme nouvelle valeur à `udp_port_rover`.

    La fonction en ligne `udp_lport_inuse()` permet de savoir si un numéro de port est déjà utilisé. Elle est définie dans le fichier `linux/include/net/udp.h` :

Code Linux 2.6.10

```
43 static inline int udp_lport_inuse(u16 num)
44 {
45         struct sock *sk;
46         struct hlist_node *node;
47
48         sk_for_each(sk, node, &udp_hash[num & (UDP_HTABLE_SIZE - 1)])
49                 if (inet_sk(sk)->num == num)
50                         return 1;
51         return 0;
52 }
```

  - Sinon on essaie de lui attribuer le numéro passé en argument après vérification. Si on n'y parvient pas, on renvoie 1.

– Dans les deux cas, on indique ce numéro de port dans l'option Internet.

La fonction en ligne `sk_unhashed()` est définie dans le fichier `linux/include/net/-sock.h`: Code Linux 2.6.10

```
288 static inline int sk_unhashed(struct sock *sk)
289 {
290         return hlist_unhashed(&sk->sk_node);
291 }
```

La fonction en ligne `sk_add_node()` est définie dans le fichier `linux/include/net/-sock.h`: Code Linux 2.6.10

```
349 static __inline__ void __sk_add_node(struct sock *sk, struct hlist_head *list)
350 {
351         hlist_add_head(&sk->sk_node, list);
352 }
353
354 static __inline__ void sk_add_node(struct sock *sk, struct hlist_head *list)
355 {
356         sock_hold(sk);
357         __sk_add_node(sk, list);
358 }
```

La fonction en ligne `sock_prot_inc_use()` est définie dans le fichier `linux/include/-net/sock.h`: Code Linux 2.6.10

```
594 /* Appelee avec les bh locales inhibees */
595 static __inline__ void sock_prot_inc_use(struct proto *prot)
596 {
597         prot->stats[smp_processor_id()].inuse++;
598 }
```

– On déverrouille en écriture et on renvoie 0.